AD-A054 098 MITRE CORP BEDFORD MASS F/G 9/2
SECURITY KERNEL VERIFICATION TECHNIQUES: ALGORITHMIC REPRESENTA--ETC(U)
APR 78 D K KALLMAN, J K MILLEN F19628-77-C-0001
UNCLASSIFIED MTR-3289 ESD-TR-78-123 NL

END
DATE FILMED
6 -78
DDC

ESD-TR-78-123

MTR-3289

SECURITY KERNEL VERIFICATION TECHNIQUES: ALGORITHMIC REPRESENTATION

BY D.K. KALLMAN AND J.K. MILLEN

APRIL 1978

Prepared for

DEPUTY FOR TECHNICAL OPERATIONS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts

Project No. 5220
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-77-C-0001

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

William R Price

WILLIAM R. PRICE, Capt, USAF
Project Engineer

John T Holland

JOHN T. HOLLAND, Lt Col, USAF
Techniques Engineering Division

FOR THE COMMANDER

Stanley P Dereska

STANLEY P. DERESKA, Colonel, USAF
Computer Systems Engineering Directorate
Deputy for Technical Operations

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ESD-TR-78-123	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: SECURITY KERNEL VERIFICATION TECHNIQUES: ALGORITHMIC REPRESENTATION	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER: MTR-3289
7. AUTHOR(s): D.K. Kallman, J.K. Millen	8. CONTRACT OR GRANT NUMBER(s): F19628-77-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The MITRE Corp., Box 208, Bedford, MA 01730	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Project No. 5220
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: APR 78 / 13. NUMBER OF PAGES: 36
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Deputy for Technical Operations, Electronic Systems Division, AFSC, Hanscom Air Force Base, MA 01731	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

FORMAL SPECIFICATIONS
LEVELS OF ABSTRACTION
SECURITY KERNEL
VERIFICATION MULTILEVEL SECURITY

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A security kernel is a hardware and software mechanism that enforces access controls to information within a computer system. Given a formal specification for a kernel, this report shows how to construct an algorithmic or programming language representation of the kernel. A technique is suggested for proving that the algorithmic representation exhibits the specified user-visible behavior. A sequence of lower levels of specification, formalizing the levels of abstraction of Dijkstra, is essential to the construction and proof technique.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

235 050

ACKNOWLEDGMENTS

This report has been prepared by The MITRE Corporation under Project No. 5220. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

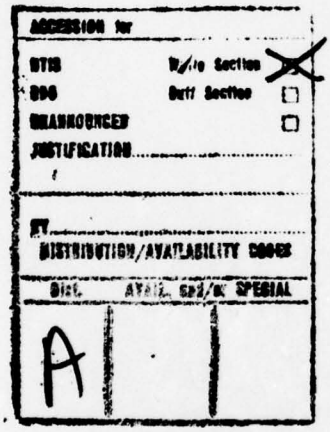

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

SECTION I

INTRODUCTION

One of the steps in the ESD/MITRE computer security program is the certification of computer systems as secure. This topic was addressed in [1] in broad terms, introducing the notion of a chain of correspondences between successive representations of a computer system. Such a chain begins with a mathematical model of secure systems and proceeds through a formal specification of the system to an algorithmic representation and finally to a machine language version.

In [2], a methodology was developed to prove that a formal specification represents a secure design. The nature and role of the formal specification of the system was explored. Using the design of a security kernel for the PDP-11/45 given in [3] as a basis, the need for multiple levels of formal specification was explained. Finally, the validation for one PDP-11/45 security kernel function at the highest level of formal specification was presented as an example of the application of the methodology.

In this report, the methodology is extended to lower levels of formal specification. The notions of levels of formal specification, and correctness of an implementation, as described by SRI in [4], are particularized to our context. This report shows how to construct an algorithmic representation from the abstract programs arising from the use of the SRI technique.

SECTION II

BACKGROUND

For the past four years, MITRE has been working on a project, sponsored by the Electronic Systems Division of the U.S. Air Force, to develop provably effective access controls for data in computer systems. Of the technical areas explored in this project, one of the most formidable has been the technical certification of multiprogrammed systems that permit controlled sharing of classified information.

Security threats to computers operating in a single-level mode are fairly well understood. Among them are physical damage, various forms of eavesdropping, and misbehavior of trusted personnel. A multilevel system is subject to an additional kind of threat, from untrusted or partially trusted users with legitimate access to the system services. Such users can introduce their own software into the system, and that software might succeed in copying classified information from one place to another, within the system, that is accessible to uncleared users.

A conceptual approach to preventing the misuse of a multilevel computer system was presented plainly in an ESD study [5]. The idea was to present all system users, and their software, with a "black box" that mediates all access to data stored in the system, and which itself could not be modified by any user. This was referred to as a reference monitor.

The reference monitor refuses all attempts at unauthorized accesses. Any requests to change access authorization also have to be processed by the reference monitor. To the user software, a reference monitor may have the functional appearance of any computer, except that ordinary instructions refuse to carry out unauthorized memory accesses, and there are additional "instructions" having to do with authorization changes.

It was recognized that rules for determining authorization and for handling authorization changes would have to be formulated and justified rigorously. Rules were sought that were consistent with the DoD policies for controlling access to classified documents. An early, independent effort toward this end was by Weissman [6] for the ADEPT-50 system. That and other work ([7], [8]) influenced the Bell-LaPadula model developed at MITRE [9].

Given a model, it was by no means clear how to implement it to produce a reference monitor. The uncertainty about what exact form the implementation should take, as well as the origins of the data protection problem in large multiprogramming and time-sharing systems, led to the approach of implementing the bulk of the reference monitor in software. The hardware and software implementing the reference monitor are called the kernel. In the context of a particular machine, the reference monitor software alone may be called the kernel [10].

In order to crystallize the problems involved in getting from a model to a kernel, MITRE undertook to implement a kernel on a minicomputer that was provided with adequate hardware protection features: a PDP-11/45 with Memory Management Unit (for segmentation with access mode control). The lessons learned from this brassboard design were then to be applied to the design of a kernel for the Multics time-sharing system. The PDP-11/45 kernel is running, in an experimental environment, and the design of the Multics kernel is underway.

The features for data protection available on a particular machine determine whether an efficient reference monitor can be implemented on it. The boundary between hardware and software responsibilities in the implementation of a reference monitor is also hardware dependent. Some discussion of what hardware features are desirable may be found in [11].

Our implementation approach can be outlined as a progression

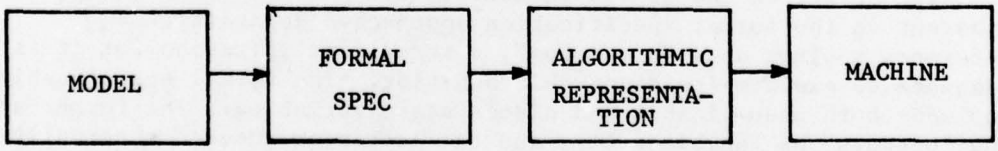

Figure 1. Stages of Kernel Representation

along four stages of kernel representation as shown in Figure 1, from Burke [1].

The formal specification embodies the design precisely, in order to verify that it satisfies the requirements of the model, and to

provide assertions against which the correctness of the software could be verified. The specification technique of Parnas [12] was adopted. The practicality of using it for a significant part of an operating system had been established by Price [13]. A formal specification for a PDP-11/45 kernel was developed from the model through the interaction of design ideas with the need for rigorous verification. The initial specification in Schiller [14] was influenced by the Multics protection mechanism ([15], [16]). Subsequent work streamlined the specification to a point where the model requirements could be verified.

At Case Western Reserve University a group took an alternative "successive refinement" approach with their model, toward the goal of a formal specification of the Multics kernel [17]. They constructed a series of models, each one a valid interpretation of the preceding one, and including more of the details necessary for an implementable specification. Their first model was influenced by and influenced the MITRE model.

VERIFICATION OF THE KERNEL DESIGN

A major point of concern, in verifying the correspondence of a kernel design to a model for protection of data, is whether all repositories of information have been identified. Initial attempts at rigorous kernel verification failed to recognize the state variables in the kernel as "objects", i.e., repositories of information. Ironically, the need to check internal variables had already been recognized in an informal analysis of security enhancements for Multics [18].

The answer to the object identification problem was already inherent in the formal specification approach. Regarding the reference monitor as a "black box", a formal specification for it is adequate to explain its observable behavior. The formal specification includes both user-visible and hidden state variables. The former are the "outputs" of the black box, and the latter are needed internally to hold additional current state information.

It suffices to identify all state variables in the specification as objects. Any implementation of the reference monitor will be constrained to have the same observable behavior with regard to user-visible state variable values. Even if its internal structure is quite different, with different hidden state variables, there should be no way that the user can tell the difference between the implementation he is using and the black box specified. If the black box is secure with the formally specified structure, it will be secure with any other structure producing the same user-observable behavior.

A real implementation may produce certain types of behavior other than the values of designated state variables. Response times, electromagnetic radiation, CPU panel lights, disk head actions, and power consumption are all potential channels for compromise of information. It is not practical to verify and implement a formal specification with all such emanations either eliminated or explicitly controlled. Methods for identifying and controlling significant channels of this type are beyond the scope of this report.

A technique for carrying out an analysis of the formal specification, with all state variables identified as objects, was described in Millen [2]. The main consequences of this technique are (1) the use of a set of axioms as the proper formal statement of authorization and access control policy, rather than rules for handling particular requests; and (2) the need for an austere version of the formal specification, sufficient to explain the reference monitor behavior, but devoid of premature implementation decisions (actually, a certain amount of unnecessary detail can be tolerated).

THE ALGORITHMIC REPRESENTATION

The algorithmic representation is a programming language version of the kernel. This report focusses on the technique for proceeding from a relatively simple formal specification of the kernel to an algorithmic representation. We are using the design methodology of Robinson, et al. [4] developed at Stanford Research Institute. SRI's technique is discussed in the next section.

Like SRI, we develop the specification by reexpressing it in lower levels of abstraction, using so-called abstract programs to show the implementation of each level in terms of the primitive functions of the next lower level.

The abstract programs between all levels are gathered together to form the major part of the algorithmic representation. The algorithmic representation also includes mapping functions or subroutines that provide the user with the black-box outputs specified at the user interface level.

SECTION III

THE SRI METHODOLOGY

The SRI methodology begins conceptually (but not necessarily chronologically) with a formal specification of a user interface. The formal specification, written in a notation derived from Parnas [12], has V-functions to store state information and O-functions to specify transformations on V-function values resulting from user requests. Such specifications may be called transition specifications.

A formal specification defines an abstract machine whose state variables are the V-function elements, whose inputs are O-function calls, and whose transitions are determined by the transformations or "effects" specified for the O-functions. An abstract machine could be a real computer if the O-functions happen to correspond to machine instructions, but the abstract machine at the user interface is more likely to be a kind of "virtual" machine.

More details of the implementation strategy are introduced, one by one: memory management, process swapping register allocation, data structure, etc. Each implementation concept yields a level of abstraction (see [19]), with its own primitives, defining its own abstract machine or a functional area within an abstract machine. Each machine is used to implement a higher level machine, just as a computer is used to implement a subroutine. To get an idea of how many levels this might involve, note that, in their work on a particular operating system design [4], SRI envisioned thirteen levels of abstraction.

The implementation of each machine using the next lower machine was specified by writing an abstract program for each O-function in the "upper" level machine in terms of "lower" level primitives. Abstract programs are ordinary programs, in some unspecified but simple and well-understood programming language. Abstract programs call O-functions instead of subroutines.

An abstract program can be either correct or incorrect depending on what assumptions have been made about the correspondence between data structures in a level and their implementation in the next lower level. SRI has chosen to require mappings that allow one to determine all the state information in each abstract machine from state information available in the next lower level machine. An abstract program is correct if it preserves the mapping between the two adjacent levels. This commutativity condition is illustrated in

Figure 2, where f is the mapping, t is the upper level O-function call, and A(t,s) is the abstract program implementing t. The commutativity condition is the equation

$$t(f(s)) = f(A(t,s)(s)).$$

This form of the condition is appropriate when f is a function; more generally, f need only be a relation. The commutativity condition is proved using well known techniques for proving assertions about programs [20]. The "input" assertion for the program A(t,s) is whatever is known about state s, and the "output" assertion is the statement that the new lower level state maps to the upper level image state.

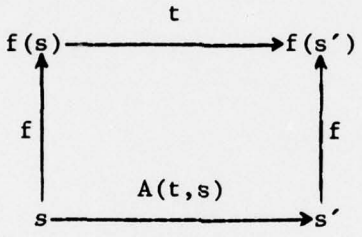

Figure 2. Correctness of Implementation

The proof methodology, called behavioral implementation, is defined in Section V. It is a combination of the more general notion of behavioral, or black box, equivalence, defined in Section IV, and the technique of abstract implementations.

Here is a brief outline of the steps proposed in this report to construct a secure algorithmic representation of the kernel:

1. Prove that the upper level of specification is secure. The methodology needed to do this is detailed in [2].

2. Design lower levels of specification.

3. For each level of specification (except the lowest) write abstract programs to implement each O-function in the next lower level.

4. Prove that each level is behaviorally implemented by the next lower level, generating invariants (relations) as needed in the proof. A relation asserting the identity of visible state variable values must be included.

5. Prove that the specific initial state chosen for the kernel satisfies the relations.

6. Write and verify function subprograms to calculate the values of high level state variables appearing in abstract programs.

7. The set of all the abstract programs, with O-function calls regarded as subroutine calls, plus the function subprograms from step 6, yield an algorithmic representation of the kernel.

SECTION IV

BEHAVIORAL EQUIVALENCE

This section discusses a method for proving the equivalence between two abstract machines, called behavioral equivalence. It is applied to show that if we have a secure upper level machine and a lower level machine that is behaviorally equivalent to it, then the lower level machine is secure. In this section we will describe behavioral equivalence for machines with the same set of inputs. A later section will show how to apply the theory when one machine (the lower level one) has a more primitive set of inputs.

DEFINITION OF BEHAVIORAL EQUIVALENCE

In order to define behavioral equivalence we define an abstract machine as a quadruple (S,s0,T,V). S is the set of states of the machine. We assume each state is a vector of state variables. s0 is the initial state of the machine. T is the set of inputs. These will be O-function calls. Not shown explicitly is the transition function taking each state s to the next state st, for each t in T. V is the vector of visible state variables. We denote by V(s) the vector of visible state variable values for a state s. In a machine defined by a Parnas specification, visible state variables are the user-visible V-function references. The user may be able to deduce information about the hidden state variables, but he can only do so through his observation of the visible state variables.

Say we are given two different abstract machines, of which one is more complicated than the other, i.e., is "lower in level". Assume that we would like to prove that the lower level machine provides a secure interface to its users. Specifically, we would like to know that, starting with a fixed initial lower level state, s0, and an arbitrary sequence of upper level inputs t1,...,tn, no information is ever passed from a user-visible state variable of a given security level to one of a lower security level during the sequence s0, s0t1,...,s0(t1..tn) of states in the lower level machine.

A way to verify security for a high level formal specification was given in [2]. That method would be impractical for a low level specification. The other machine will be secure, however, if it exhibits the same user-visible behavior as one that has been proved to be secure already. This approach works if the two machines have the same set of inputs and the same user-visible state variables, as we are assuming in this section.

Formally, two machines (S,s0,T,V) and (S′,s0′,T,V) are defined to be behaviorally equivalent if, for every sequence of inputs, t1,...,tn,

$$V(s0(t1...tn)) = V\ (s0'(t1...tn)).$$

This means that both machines produce the same outputs (values on visible state variables) for all time starting with the given initial states.

Pictorially, behavioral equivalence is:

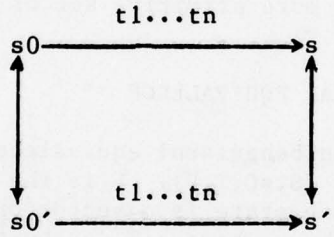

Figure 3. Behavioral Equivalence

where ⟷ means that the two states have the same values on visible V-functions.

PROOF TECHNIQUE

The definition of behavioral equivalence yields naturally to an inductive proof technique. We will call the relation V(s) = V (s′) the behavioral equivalence relation (BE). It would suffice to show that:

1. The pair of initial states s0, s0′ satisfies BE, and

2. If (s, s′) satisfies BE then for all inputs t in T, (st, st′) satisfies BE,

that is, each possible state transition preserves BE.

As often happens in inductive proofs, the induction hypothesis has to be stronger than the statement to be proved. We will usually need additional relations. Many additional relations arise from the

fact that visible state variables are often references to derived V-functions. Consequently, BE must be supported by relations among the hidden primitive V-functions that define the visible, derived ones. Furthermore, both levels individually must work "properly", and this means that a number of relations that are fundamental assumptions in the design are needed. These relations insure that we consider only "consistent" states.

The proof starts with BE alone, but the additional relations suggest themselves during the proof. The technique of generating relations as needed is familiar from the top level to model validation, where we started with relations expressing the axioms and generated several more relations embodying design invariants.

SECTION V

BEHAVIORAL IMPLEMENTATION

In defining behavioral equivalence, it was convenient to assume that both machines had the same set of inputs and the same set of visible state variables. The required state variables can always be added to a machine described by a lower level specification, as derived V-functions. Matching the inputs is harder. In a multilevel specification, the transitions resulting from inputs in a lower level are often more primitive that those in higher levels. It may take a long sequence of lower level inputs to produce the same effect as one upper level input.

Given the machine described by a lower level specification, we will construct a new algorithmic machine that accepts the same inputs as the upper level machine. The transition caused by an upper level input is the net effect of the corresponding sequence of lower level transitions. This new algorithmic machine can then be compared directly with the higher level machine for behavioral equivalence. The algorithmic machine has the same inputs and visible state variables as the high level machine, but has the hidden state variables of the lower level machine.

A mechanism for translating upper level inputs in this fashion is already available: abstract programs. An abstract program translates an upper level input into a sequence of lower level inputs.

An abstract program, as described by SRI, uses the functions of a lower level specification and the control constructs of some formally defined programming language. While the details of the control constructs have not been specified by SRI and will not be specified here, we can imagine that a small, simple set like succession, if-then, and while-do, are used. Some remarks on other likely characteristics of an abstract programming language will appear later in this section and in Section VII.

The construction of a new algorithmic machine from a lower level specification and a collection of abstract programs, one for each upper level O-function, is illustrated in Figure 4. The figure points up a flaw in the use of more primitive machines to implement a virtual environment: a single input will produce a sequence of changes in visible state variables, rather than the single change specified in the top level. The sequence of changes can obviously transmit more information than the net change.

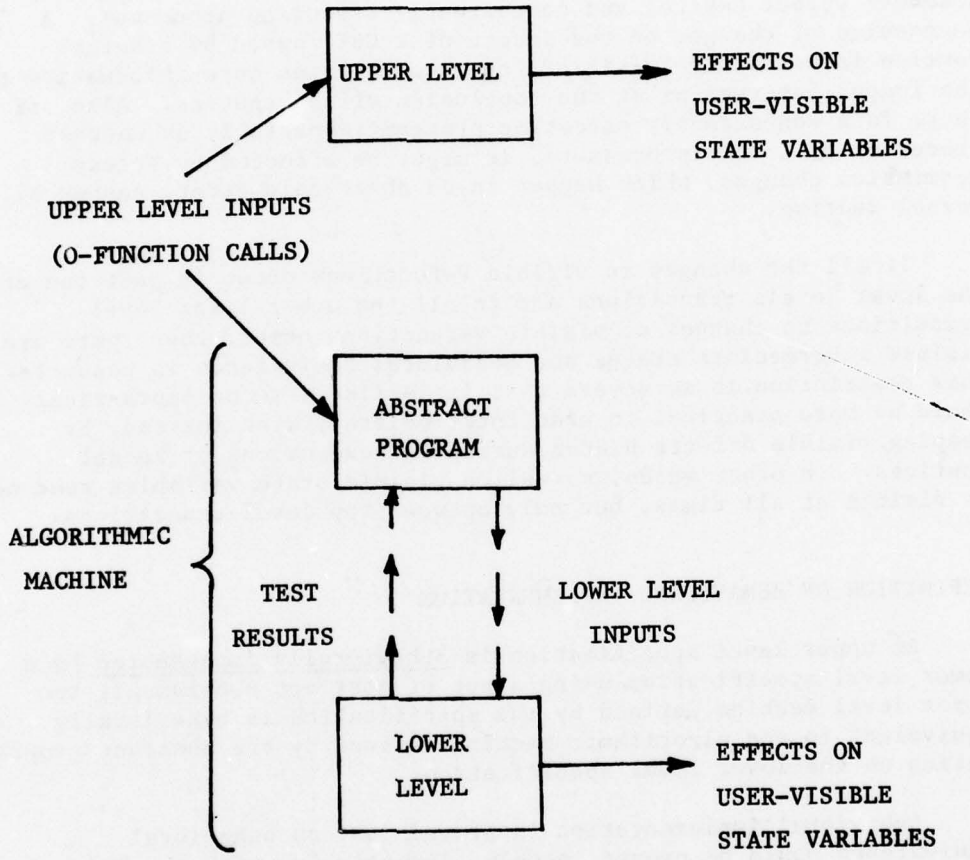

Figure 4. Behavioral Implementation

The seriousness of this intermediate state problem depends on the real-world implementation of the kernel. Usually kernel routines are not interruptible by their caller, and any changes they make in memory contents would not be observable to the calling program until the kernel has returned control to it. There are still two danger spots, however: output devices and concurrently executing processes. A succession of changes on the screen of a CRT caused by a kernel routine is certainly "visible", and communicates more information than the image that remains at the conclusion of the routine. Also, if there is a concurrently executing program (especially on another processor of a multiprocessor), it might be affected by access permission changes, which happen in an observable order, caused by a kernel routine.

If all the changes in visible V-functions occur in just one of the lower levels transitions and in all the other lower level transitions no changes on visible V-functions occur, then there are no visible intermediate states and behavioral equivalence is possible. This restriction is so severe that it is likely to be impractical. It would be more practical to hide intermediate states instead, by keeping visible effects hidden during the executions of kernel routines. In other words, so-called visible state variables need not be visible at all times, but only between top level transitions.

DEFINITION OF BEHAVIORAL IMPLEMENTATION

An upper level specification is behaviorally implemented by a lower level specification, using a set of abstract programs, if the upper level machine defined by the specification is behaviorally equivalent to the algorithmic machine defined by the abstract programs acting on the lower level specification.

Behavioral implementation is proved just as behavioral equivalence would be proved, namely, inductively, with the help of a sufficiently large set of relations. Showing that a relation is preserved by the net effect of an abstract program is a special case of the task of showing that a program entered with an input assertion true finishes with an output assertion true. (See, for example, [21].) Each relation serves as both an input and an output assertion.

An abstract implementation associates an abstract program with each upper level input. When the program is entered in a particular lower level state, the result is a path of execution through the program, determined by tests on lower level state variables, yielding a sequence of lower level inputs.

Formally, let M and M′ be machines, where M = (S,T,s0,V) and M′ = (S′,T′,s0′,V′). An abstract implementation is a function:

$$A: S' \times T \rightarrow T'^{*},$$

where T′* is the set of finite sequences of lower level inputs. Then M is <u>behaviorally implemented</u> by M′, using A, with respect to a relation R contained in SxS′ if

1. R implies BE (i.e., if (s,s′) is in R then V(s) = V′(s′)),

2. (s0,s0′) is in R (i.e., the initial states are R-related), and

3. if (s,s′) is in R then (st,s′A(s′,t)) is in R (R is preserved by upper level inputs).

In practice, the relation R will be a conjunction of several relations on state variables, including BE.

REMARKS ON SYNTAX OF ABSTRACT PROGRAMS

The fundamental requirements for abstract programs are:

1. All effects on lower level state variables result from lower level O-function calls.

2. The sequence of lower level O-function calls is determined entirely by the upper level O-function call (input) parameters and the lower level state variables.

The abstract program format, as an algorithm written in a programming language, is just a constructive way of defining the sequence of lower level O-function calls as required by (2).

Because program proofs are necessary to show behavioral implementation, the semantics of the abstract programming language must be rigorously defined. Limitations in the choice of control structures, data types, and mathematical operations may have to be observed if program verification tools are to be used.

A question that comes up often in defining the semantics of abstract programs is what to do when a called O-function returns with an exception condition. For Parnas specifications, the desired effect

of an error trap is a transfer of control to a named error handling routine. We may treat a sequence of exceptions x1, x2,... as if it had been written as part of the effect: if x1 then EC = 1 else if x2 then EC = 2 ... else EC = 0, where EC is an auxiliary state variable used as an error code. Control returns to the abstract program in the usual place after the O-function call, and it becomes the responsibility of the abstract program to test the value of EC and adjust its flow of control accordingly. This will be referred to as the error code approach.

An expression of the form "if a then b else if c then d else e" in a Parnas effect section is just an abbreviation for the logical expression a&b or (not a)&c&d or (not a)&(not c)&e. The value of this expression gives the same information as if the tests on a, c,... had been carried out in the indicated order. Thus, in writing actual specifications, this treatment of exceptions can be stipulated for verification purposes while the exceptions are simply listed in the usual way without the if - then connectives.

Abstract programs may be expected to have program variables and assignment statements. Assignments should have program variables on the left-hand side. These variables should not be lower level state variables, since only O-function calls should modify state variables. Arithmetic or boolean expressions, using program variables and all state variables can appear on the right-hand side of assignment statements.

Abstract programs do not need subroutine calls, unless recursion is their iteration method. Any temptation to use subroutines probably indicates a need for another level of specification between the upper and lower levels, since subroutines usually express some sort of primitive operations.

SUMMARY

In this section we introduced the notion of behavioral implementation in order to extend behavioral equivalence to machines that do not have the same inputs. In behavioral implementation each input of the first machine is implemented by a sequence of inputs on the second machine. One must prove that the sequence of inputs on the second machine produces the same user visible effect as the original input on the first machine.

Abstract programs express the implementation of the inputs of the first machine by the second. They are written in terms of the variables and O-functions of the second, lower level machine, plus program variables. They call O-functions for any modification of the lower level state variables.

SECTION VI

MULTILEVEL BEHAVIORAL IMPLEMENTATION

INTRODUCTION

To this point our theory has revolved around only two levels of specification. We have seen:

1. How to prove that a lower level behaviorally implements an upper level.

2. If a lower level behaviorally implements a secure upper level, the algorithmic machine obtained by using the abstract implementations of the upper level inputs in the lower level is secure.

We have also seen, in our discussion of the SRI design methodology, that we would like to decompose the design of the kernel into more than two levels. In order to apply the theory for two levels to multi-level specifications it would help to know when the implementation relationship is transitive. That is, if M" implements M′ with the abstract implementation A, and M′ implements M with A', can we construct, from A′ and A, an abstract implementation A" of M by M"? We will show how the construction can be carried out when a certain condition relating successive implementations is satisfied.

ABSTRACT CONSTRUCTION

Consider the following situation, illustrated in Figure 5.

Hypothesis 1. We have three levels of specification, the upper, middle and lower levels, defining machines M, M′, and M".

Hypothesis 2. The upper level is behaviorally implemented by the middle w.r.t., a relation set R. This means that for all upper level inputs t, and for all middle level states, s′, there is a sequence of middle

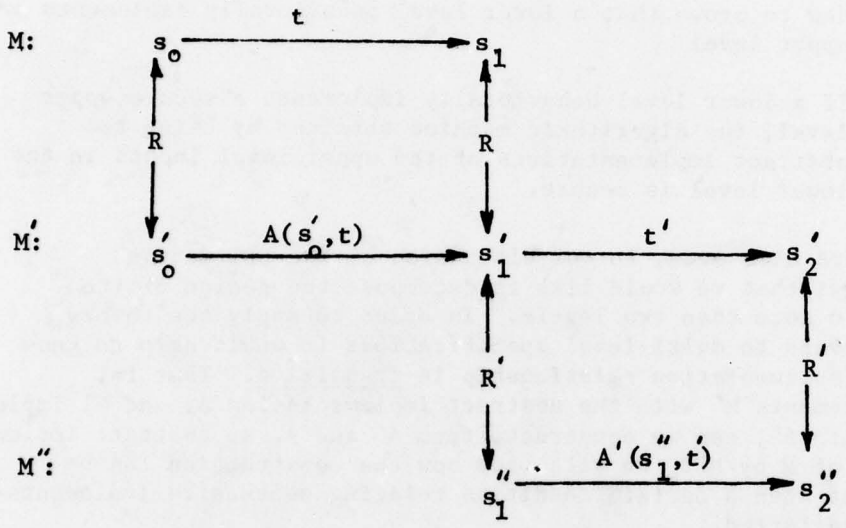

Figure 5. Three Levels

level inputs, denoted A(s′,t), such that for all (s,s′) in R, (st,s′A(s′,t)) is in R.

Hypothesis 3. The middle level is behaviorally implemented by the lower level w.r.t., the relation set R′. The abstract implementation is denoted A′. We have for all (s′,s") in R′ and all t′ in T′, (s′t′,s"A′(s",t′)) is in R′.

We would like to prove that the upper level is behaviorally implemented by the lower level with respect to some relation set, R".

1. We let R" = R∘R′ = {(s, s") in SxS"|∃s′ in S′ such that (s,s′) is in R and (s′, s") is in R′}.

2. We must now construct A". We note the following:

   a. A′:S"xT′ -> T"* can be extended in a natural way to a map S"xT′* -> T"*, also denoted A′. This is done recursively by defining

      $$A'(s'',t't\hat{}) = A'(s''A'(s'',t'),t\hat{})$$

      for t^ in T′*.

   b. If (s′,s") is in R′ and t^ is in T′* then (s′t^, s"A′(s",t^)) is in R′. This can be proved inductively on the length of t^.

To define A" we need the following additional hypothesis:

Hypothesis 4 (Well-Definition, W-D). If (s1′,s") and (s2′,s") are in R′ then A(s1′t) = A(s2′,t) (See Figure 6).

We now define A":S"xT -> T"* via:

   c. If there is s′ in S′ such that (s′,s") is in R′ then A"(s",t) = A′(s",A(s′,t)).

      We note that by the Well-Definition hypothesis, this definition is invariant of the choice of s′.

We will see below that A" need not be defined when s′ does not exist.

Theorem. If hypotheses 1-4 are true then the lower level behaviorally implements the upper level w.r.t., the relation set R" using the abstract implementation A".

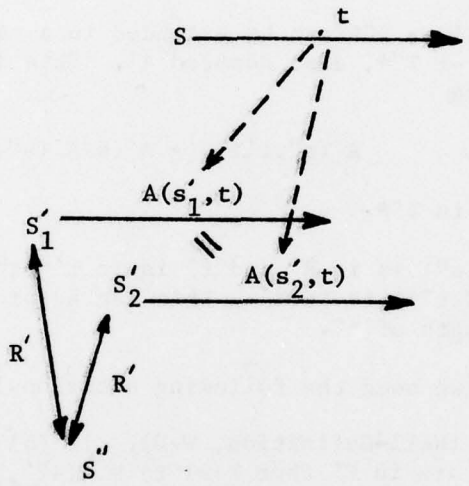

Figure 6. The Well-Definition Hypothesis

Proof: Fix (s',s") in R" and t in T. We must show (st,s"A"(s",t)) is in R". But by definition of R" there exists s' in S' such that (s,s') is in R and (s',s") is in R'.

(s,s') is in R implies (st,s'A(s',t)) is in R.

By definition of A",

$$A"(s",t) = A'(s",A(s',t)).$$

Therefore, $s"A"(s",t) = s"A'(s",A(s',t))$. (*)

By remark 2b) above, letting $t^{\wedge} = A(s',t)$, we have $(s'A(s',t),s"A'(s",A(s',t)))$ is in R'.

By (*) we have $(s'A(s',t),s"A"(s",t))$ is in R'.

Now we have $(st,s'A(s',t))$ is in R and $(s'A(s',t),s"A"(s",t))$ is in R'. Therefore $(st,s"A"(s",t))$ is in R", as desired.

EXPLANATION OF WELL-DEFINITION HYPOTHESIS

We saw in the previous section that in order to prove transitivity of behavioral equivalence we needed the additional hypothesis:

W-D: if (s1',s") and (s2',s") are in R' then A(s1',t) = A(s2',t).

This hypothesis was needed in order to be able to construct the abstract implementation of the upper level by the lower level. The problem essentially is that the abstract implementation of the upper level by the lower level can only use lower level V-functions, since the abstract implementation of the upper level by the lower level is a map taking S"xT->T"*. The obvious method of composing the two abstract implementations A' and A still leaves us with middle level V-functions.

For example, let t be an upper level input, v' a middle level state variable, t' a middle level input, and t" a lower level input. If our abstract programs (representing the abstract implementations) are defined as follows (where "call" followed by an input means apply that input to the appropriate machine):

A(s',t) = if v' then call t'

A'(s",t') = call t",

we would expect to be able to define the implementation of t in the lower level as:

A"(s",t) = if v′ then call t".

But this is not allowed. We have a middle level V-function present in the abstract program of the upper level by the lower level. In general, we just cannot get rid of it. Suppose, however, that we have sufficiently restricted the relation set between the middle and lower levels so that for every allowable pair of middle and lower level states, the value of v′ is a function of the lower level state, denoted v′ = f(s"). We can then write:

A"(s",t) = if f(s") then call t".

The function f can be expressed as a derived lower level V-function. The relation v′ = f is then added to the relation set determining R′, to make the W-D Hypothesis true. Now the program for A" involves only lower level V-functions, as desired.

In general, if all the middle V-functions that appear in the abstract implementation of the upper level by the middle level are functions of lower level V-functions, then we can combine the abstract programs of the upper to middle and middle to lower levels to get an abstract implementation of the upper level by the lower level involving only lower level V-functions.

It can easily be shown that if all the middle level V-functions referenced in the abstract programs are functions of lower level V-functions, then the Well-Definition Hypothesis is satisfied. Even where this condition is not true at first, it can sometimes be made true trivially by adding derived V-functions.

Consider the following example:

Let:

```
A(s′,t) = if v1′&v2′
              then call t′;
```

The value of A(s′,t) depends only on the value of v1′&v2′. It may not be the case that both of v1′, v2′ are functions of the lower level, while at the same time (v1′&v2′) is a function of the lower level.

Let us make a mapping V-function vtest in the middle level whose value is v1′&v2′. We now have:

```
A(s′,t) = if vtest
                then call t′;
```

With this representation of A, we can say that all middle level V-functions appearing in the abstract implementation are functions of the lower level, even though this was not true at first.

In practice, what we do to guarantee the existence of an abstract implementation of the upper level by the lower level is:

1. Identify the middle level state variables appearing in the abstract implementation of the upper level by the middle level, A.

2. If necessary, add derived V-functions in the middle level to make step 3. possible.

3. Express all the middle level state variables appearing in A as functions of the lower level. For each one, use a lower level derived V-function and a relation identifying it with the middle level state variable.

Note that if we construct A" as above, we now have appearing in A" lower level V-functions of two types:

1. those originally in A′

2. those needed to map to the V-functions in A.

SUMMARY

In this section the theory of behavioral implementation was developed for three or more levels. It was shown that under an assumption called the Well-Definition Hypothesis, an abstract implementation of the top level by the bottom level can be constructed from the implementations given between successive levels.

The Well-Definition hypothesis is satisfied if there are relations expressing the state variables used in abstract programs as functions of lower level state variables. Such relations are mapping relations between successive levels, like BE, and are proved inductively in the same way. The functions thus shown to exist are translated into function subprograms to form an important part of the algorithmic representation, as described in Section VII.

SECTION VII

ALGORITHMIC REPRESENTATIONS

In the last section we saw that if the relations between levels were sufficiently strong, then we could use the abstract programs between levels to mathematically construct an abstract implementation of the top level by the bottom level. This section shows how to use these abstract programs in an algorithmic representation of the kernel.

The idea is to retain the abstract programs unchanged as subroutines of the algorithmic representation. In doing so, O-function calls are viewed as subroutine calls, and references to derived functions or to higher level V-functions are viewed as calls on function subprograms. The complete algorithmic representation includes:

1. the original abstract programs between all successive levels,

2. function subprograms to calculate derived function values, and

3. function subprograms to calculate higher level primitive state variables from lower level state variables.

V-function and O-function references in the final programs must all belong to the bottom level, and the V-functions must be primitive.

EXAMPLE

Figure 7 shows three specifications. The top one is the upper level specification for a machine with one data structure, a list, and one O-function, BEHEAD, which removes the first element of the list each time it is called, until the list is empty. The visible part of the list shortens by one element each time BEHEAD is called. This visibility requirement is embodied in a derived V-function VLIST which returns zero for the elements that have become hidden. The primitive V-function LIST is entirely hidden.

The second specification describes a middle level linked-list machine on which the upper level will be implemented. Note that the two visible V-functions of the upper level, LENGTH and VLIST, also appear in the middle level, although here they are both derived. Their presence is necessary for behavioral equivalence.

UPPER

```
Hidden V-function LIST(i)

Visible V-function LENGTH

O-function BEHEAD:
   Effect: if 'LENGTH' ≠ 0
           then LENGTH = 'LENGTH' - 1
            and (∀ i) if 1 ≤ i < 'LENGTH'
                        then LIST(i) = 'LIST'(i + 1)

Visible Derived V-function VLIST(i):
        Derivation: if 1 ≤ i ≤ LENGTH then LIST(i) else 0
```

MIDDLE

```
Hidden V-function VALUE(i)

Hidden V-function PTR(i)

Hidden V-function HEAD

O-function CHOP:
   Effect: HEAD = PTR('HEAD')

Visible Derived V-function LENGTH:
        Derivation: COUNT(HEAD)

Hidden Recursive V-function COUNT(i):
       Derivation: if i = 0 then 0 else 1 + COUNT(PTR(i))

Visible Derived V-function VLIST(i):
        Derivation: VALUE(WALK(HEAD,i))

Hidden Recursive V-function WALK(j,i):
       Derivation: if i = 1 then j else WALK(PTR(j),i-1)
```

LOWER

```
Visible V-function LENGTH

Hidden V-function STACK(i)

O-function DROP:   Effect: LENGTH = 'LENGTH' - 1

Visible Derived V-function VLIST(i):
        Derivation: if 1 ≤ i ≤ LENGTH
                    then STACK(LENGTH - i + 1)
                    else 0
```

Figure 7. Example Specifications

Note also that two of the derived V-functions are defined recursively. This imposes an obligation to prove the convergence of these definitions.

A CHOP operation is provided in the middle level, and using it, the upper level operation BEHEAD can be implemented with the following abstract program:

```
Program BEHEAD:
        if HEAD ≠ 0
        then call CHOP
```

The third specification describes a stack machine which also includes the visible V-functions of the upper level. Its job is to implement the middle level O-function, for which it provides a simple O-function DROP which merely lowers the stack pointer. The abstract program for CHOP is:

```
Program CHOP:
        if LENGTH ≠ 0
        then call DROP
```

Even for such a small system, the proof of behavioral equivalence is tedious enough so that it will not be given here, nor will the proof of convergence of the two recursive functions in the middle level. In developing the algorithmic representation, however, we should be aware that at least BE is satisfied between each of the pairs of levels. Since we have given corresponding V-functions the same names, BE can be stated as:

$$\text{LENGTH} = \text{LENGTH}' = \text{LENGTH}''$$

and (BE)

$$\text{LIST}(i) = \text{LIST}'(i) = \text{LIST}''(i)$$

where one prime (′) refers to the middle level, and two to the lower.

The next step is to check the W-D Hypothesis to establish transitivity. W-D will be satisfied in particular if each middle level V-function appearing in the abstract program for BEHEAD is a function of the lower level. However, the lower level does not contain enough information to determine HEAD, the one V-function that occurs. In this case we are able to take advantage of the tactic suggested in Section VI. Since the test in the BEHEAD program is only for HEAD ≠ 0, we can define a new middle level V-function STOP which is sufficient to handle the test and which is a function of the lower level. Let STOP be specified by:

Hidden V-function STOP:
Derivation: HEAD = 0

Clearly BEHEAD can now be rewritten in terms of STOP as:

```
Program BEHEAD:
        if not STOP
        then call CHOP
```

We now have to substantiate the claim that STOP is a function of the lower level. The following relation will be proved:

$$STOP = (LENGTH'' = 0).$$

To prove this relation we observe, first, that by BE,

$$LENGTH = LENGTH'$$

so that it suffices to show that:

$$STOP = (LENGTH' = 0)$$

But $LENGTH' = COUNT(HEAD)$, and it is easily shown from the derivation of COUNT that $COUNT(i) = 0$ if and only if $i = 0$. It follows that $LENGTH' = 0$ if and only if $HEAD = 0$, as required.

Since STOP has been established as a function of the lower level, we can regard the reference to STOP in BEHEAD as a function call, and it remains only to implement a function subprogram to calculate it. The subprogram for STOP would be:

```
Function subprogram STOP:
                    if LENGTH'' = 0
                    then STOP = true
                    else STOP = false
```

Finally, we must be sure that all other derived V-functions needed for the implementation have been provided. In this case, we still need a function subprogram for VLIST, say:

```
Function subprogram VLIST(i):
                    if 1 ≤ i ≤ LENGTH''
                    then VLIST(i) = STACK(LENGTH'' - i + 1)
                    else VLIST(i) = 0
```

The two function subprograms above, plus the rewritten program for BEHEAD and the original program for CHOP, constitute the algorithmic specification for the BEHEAD system, in terms of the STACK system at the primitive level.

SUMMARY

The algorithmic specification language can be the abstract programming language, extended to permit subroutines and function subprograms. This choice of language makes the verification requirements clear cut. If an algorithmic representation in another language is to be verified, the best bet is to construct the algorithmic representation as described here, using the abstract programming language, and then prove equivalence between the two.

The algorithmic representation suggested here consists of the abstract programs already written, plus some function subprograms needed to implement mappings. If behavioral implementation has been proved as in Section V, using a relation set strong enough to guarantee transitivity as in Section VI, and the calling mechanism is correct, then only the new function subprograms need be verified to complete the proof of the algorithmic representation.

SECTION VIII

SUMMARY AND CONCLUSIONS

In this paper we have explored the techniques by which one would go from a top-level formal specification of the kernel to an algorithmic representation of the kernel. The method recommended is to write successively lower level specifications, each describing a machine closer to the hardware, and each one implemented by the next machine using abstract programs written in a simple abstract programming language with no calls except to lower level functions. The last specification should have as primitives the elementary statements and built-in subroutines of the implementation language.

Behavioral, or black box, equivalence between abstract machines was defined in terms of user-visible state variables. Behavioral implementation is the result of applying the abstract implementation methodology to the goal of behavioral equivalence.

A technique based on preserving a set of relations was outlined for the proof that each level behaviorally implements the one above. The set of relations includes the behavioral equivalence relation (BE). Abstract programs are used to express the implementation of upper level inputs via sequences of lower level inputs.

With three or more levels of specification, the abstract programs between the successive pairs of levels can be combined into a single overall implementation of the top level by the bottom level if the Well-Definition Hypothesis (W-D) is satisfied. It will be satisfied if each state variable or test appearing in an abstract program is expressible in terms of lower level state variables.

The abstract programs form the core of an algorithmic representation. Function subprograms are then added to calculate all tested values and parameters in the bottom level. The verification of these function subprograms completes the proof that the abstract machine described by the algorithmic representation is behaviorally equivalent to the top level. If the top level is proved secure with respect to a suitable set of axioms, the algorithmic machine is secure.

REFERENCES

1. E. L. Burke, "Synthesis of a Software Security System," MTP-154, The MITRE Corporation, Bedford, Massachusetts, August 1974.

2. J. K. Millen, "Security Kernel Validation in Practice," Communications of the ACM, Volume 19, Number 5, May 1976, pp. 243-250.

3. W. L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45," ESD-TR-75-69, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., May 1975 (AD A01171).

4. L. Robinson, P. G. Neumann, K. N. Levitt, and A. R. Saxena, "On Attaining Reliable Software for a Secure Operating System," 1975 International Conference on Reliable Software, Los Angeles, California, April 1975, pp. 267-284.

5. J. P. Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, Volume I and II, James P. Anderson & Co., Fort Washington, Pennsylvania, October 1972.

6. C. Weissman, "Security Controls in the ADEPT-50 Time-Sharing System," Proceedings AFIPS 1969 FJCC, AFIPS Press, Montvale, New Jersey, 1969, pp. 119-133.

7. R. R. Schell, P. J. Downey, and G. J. Popek, "Preliminary Notes on the Design of Secure Military Computer Systems," MCI-73-1, Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, Massachusetts, January 1973.

8. B. W. Lampson, "Protection," Proceedings 5th Annual Princeton Conference, Princeton, New Jersey, March 1971, pp 437-443 (reprinted in ACM Operating Systems Review, Volume 8, Number 1, January 1974, pp. 18-24).

9. D. E. Bell and L. J. LaPadula, "Secure Computer Systems," ESD-TR-73-278, Volumes I-III, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., November 1973-April 1974 (AD 770768, 771543, 780528).

10. "ESD 1974 Computer Security Developments Summary," MCI-75-1, Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, Massachusetts, December 1974.

11. L. Smith, "Architectures for Secure Computer Systems," ESD-TR-75-51, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., April 1975 (AD A009221).

12. D. L. Parnas, "A Technique for Software Module Specification with Examples," Communications of the ACM, Volume 15, Number 5, May 1972, pp. 330-336.

13. W. R. Price, "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1973.

14. W. L. Schiller, "Design of a Security Kernel for the PDP-11/45," ESD-TR-73-294, Electronic Systems Division, Hanscom AF Base, Mass., December 1973 (AD 772808).

15. J. H. Saltzer, "Protection and the Control of Information in Multics," Communications of the ACM, Volume 17, Number 7, July 1974, pp. 388-402.

16. E.I. Organick, The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Massachusetts, 1972.

17. K. G. Walter, W. F. Ogden, et. al., "Initial Structure *Specification for an Uncompromisable* Computer Security System," ESD-TR-75-82, Case Western Reserve University, Cleveland, Ohio, July 1975.

18. J. C. Whitmore, A. Bensoussan, P. A. Green, A. M. Kobziar, and J. A. Stern, "Design for Multics Security Enhancements," ESD-TR-74-176, Honeywell Information Systems, 1974.

19. E. W. Dijkstra, "The Structure of the 'THE' Multiprogramming System," Communications of the ACM, Volume 11, Number 5, May 1968, pp. 341-346.

20. R. W. Floyd, "Assigning Meaning to Programs," Mathematical Aspects of Computer Science, Volume 19, American Mathematics Society, Providence, Rhode Island, 1967, pp. 19-32.

21. C. A. R. Hoare, "Proof of a Program: FIND," Communications of the ACM, Volume 14, Number 1, January 1971, pp. 39-45.

22. International Business Machines, IBM System/370 Principles of Operation, GA22-7000-4, September 1974.

23. Digital Equipment Corporation, PDP-11/45 Processor Handbook, 1973.

24. Multics Programmers' Manual, AG91 revision 1, AG92C revision 1, AG93C revision 1, and AK92 revision 1, Honeywell Information Systems Inc., July 1976.